**22AIE203 –**

**Data structures and algorithms -2**

**Lab 5**

Name : Anuvind M P

Roll : AM.EN.U4AIE22010

**Sparse Matrix :**

class SparseMatrix:

     def \_\_init\_\_(self, rows, cols):

          self.\_\_data = [[rows, cols, 0]]

     def \_\_setitem\_\_(self, key, value):

          if (key[0] > self.\_\_data[0][0]-1) or (key[1] > self.\_\_data[0][1]-1):

               print(f"Invalid Indexing: {key} for size of {tuple(self.\_\_data[0][:2])}")

          if self[key] == 0:

               self.\_\_data[0][2]+=1

               self.\_\_data.append([key[0], key[1], value])

               header = self.\_\_data[0]

               data = sorted(self.\_\_data[1:])

               self.\_\_data = [header]

               for i in data:

                    self.\_\_data.append(i)

          else:

               header = self.\_\_data[0]

               self.\_\_data = [[i[0], i[1], value] if tuple(i[:2]) == key else i for i in self.\_\_data[1:]]

               self.\_\_data.insert(0, header)

     def \_\_getitem\_\_(self, key):

          if (key[0] > self.\_\_data[0][0]-1) or (key[1] > self.\_\_data[0][1]-1):

               print(f"Invalid Indexing: {key} for size of {(self.\_\_data[0][0]-1, self.\_\_data[0][1]-1)}")

               return None

          for i in self.\_\_data[1:]:

               if (i[0], i[1]) == key:

                    return i[2]

          return 0

     def \_\_add\_\_(self, other):

          if self.\_\_data[0][:2] == other.\_\_data[0][:2]:

               header = self.\_\_data[0]

               newinstance = SparseMatrix(header[0], header[1])

               for i in self.\_\_data[1:]:

                    newinstance[(i[0], i[1])] = i[2]

               for i in other.\_\_data[1:]:

                    newinstance[(i[0], i[1])] = newinstance[(i[0], i[1])] + i[2]

               return newinstance

          else:

               print(f"Dimension Error: the Dimensions ({self.\_\_data[0][0]}, {self.\_\_data[0][1]}) and ({other.\_\_data[0][0]}, {other.\_\_data[0][1]}) doesn't match")

     def \_\_str\_\_(self):

          res=""

          data = [[0 for i in range(self.\_\_data[0][1])] for i in range(self.\_\_data[0][0])]

          for i in self.\_\_data[1:]:

               data[i[0]][i[1]] = i[2]

          for i in data:

               for j in i:

                    res+="{:>5}".format(round(j, 2))

               res+="\n"

          return res[:-1]

     def display(self):

          print("+-----+-----+-----+-----+")

          print("| Ind | Row | Col | Val |")

          print("+-----+-----+-----+-----+")

          for i, val in enumerate(self.\_\_data):

               if i==0:

                    print("| {:<3} | {:<3} | {:<3} | {:<3} |".format(i+1, val[0], val[1], val[2]))

                    print("+-----+-----+-----+-----+")

                    continue

               print("| {:<3} | {:<3} | {:<3} | {:<3} |".format(i+1, val[0]+1, val[1]+1, val[2]))

          print("+-----+-----+-----+-----+")

     def transpose(self):

          for i in self.\_\_data:

               i[0], i[1] = i[1], i[0]

     def inverse(self):

          if self.\_\_data[0][0] != self.\_\_data[0][1]:

               print("Inverse Error: The Matrix is not a square matrix")

               return

          if self.determinant() == 0:

               print("Inverse Error: The Matrix is Indefinite Matrix")

               return

          matrix = [[0 for \_ in range(self.\_\_data[0][1])] for \_ in range(self.\_\_data[0][0])]

          for i in self.\_\_data[1:]:

               matrix[i[0]][i[1]] = i[2]

          n = len(matrix)

          identity = [[0] \* n for \_ in range(n)]

          for i in range(n):

             identity[i][i] = 1

          # Gaussian Inverse

          for i in range(n):

               # Partial pivoting

               max\_row = i

               for j in range(i + 1, n):

                    if abs(matrix[j][i]) > abs(matrix[max\_row][i]):

                         max\_row = j

               matrix[i], matrix[max\_row] = matrix[max\_row], matrix[i]

               identity[i], identity[max\_row] = identity[max\_row], identity[i]

               scalar = 1.0 / matrix[i][i]

               for j in range(n):

                    matrix[i][j] \*= scalar

                    identity[i][j] \*= scalar

               for j in range(n):

                    if i != j:

                         scalar = matrix[j][i]

                         for k in range(n):

                              matrix[j][k] -= scalar \* matrix[i][k]

                              identity[j][k] -= scalar \* identity[i][k]

          self.\_\_data = [self.\_\_data[0]]

          self.\_\_data[0][2] = 0

          for i in range(self.\_\_data[0][0]):

               for j in range(self.\_\_data[0][1]):

                    if identity[i][j] != 0:

                         self[(i, j)] = identity[i][j]

     def determinant(self):

          if self.\_\_data[0][0] != self.\_\_data[0][1]:

               print("Determinant Error: The matrix is not square")

               return None

          matrix = [[0 for \_ in range(self.\_\_data[0][1])] for \_ in range(self.\_\_data[0][0])]

          for i in self.\_\_data[1:]:

               matrix[i[0]][i[1]] = i[2]

          det = 1

          for i in range(self.\_\_data[0][0]):

               maxElem = abs(matrix[i][i])

               maxRow = i

               for k in range(i + 1, self.\_\_data[0][0]):

                    if abs(matrix[k][i]) > maxElem:

                         maxElem = abs(matrix[k][i])

                         maxRow = k

               if i != maxRow:

                    det \*= -1

                    matrix[i], matrix[maxRow] = matrix[maxRow], matrix[i]

               det \*= matrix[i][i]

               if matrix[i][i] == 0:

                    return 0

               for k in range(i + 1, self.\_\_data[0][0]):

                    c = 0

                    if matrix[k][i] != 0:

                         c = -matrix[k][i] / matrix[i][i]

                         for j in range(i, self.\_\_data[0][0]):

                              matrix[k][j] += c \* matrix[i][j]

          return det

sp = SparseMatrix(5, 5)

sp[(1, 2)] = 3

sp[(4, 1)] = 1

sp[(2, 0)] = 4

sp[(3, 3)] = 12

sp2 = SparseMatrix(5, 5)

sp2[(1, 2)] = 3

sp2[(4, 1)] = 1

sp2[(2, 0)] = 4

sp2[(3, 3)] = 12

print(sp)

print(sp2)

sp.display()

sp2.display()

sp3 = sp + sp2

print(sp3)

sp3.display()

print(sp2)

sp2.transpose()

print(sp2)

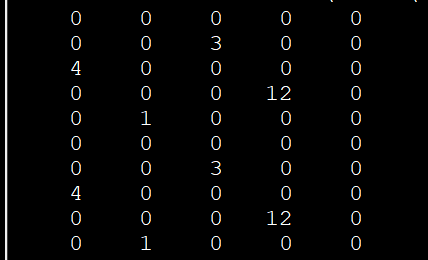
print()

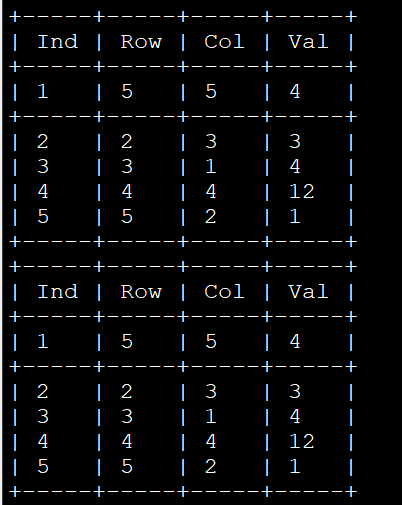
print(sp3)

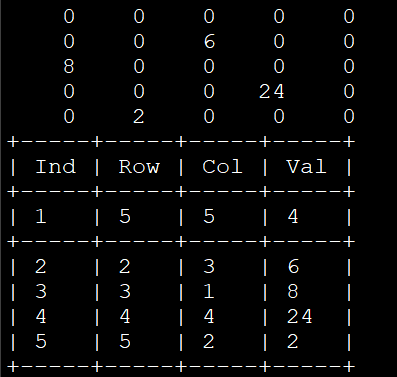
sp3.inverse()

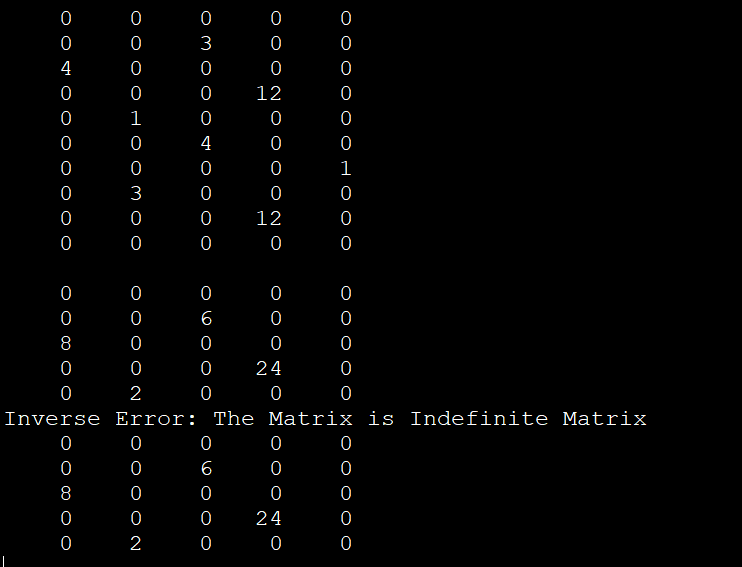
print(sp3)

**Output :**









**Block Matrix :**

class BlockMatrix:

    def \_\_init\_\_(self, size, block\_size):

        self.size = size

        self.block\_size = block\_size

        self.num\_blocks = size // block\_size

        self.blocks = [[[0] \* block\_size for \_ in range(block\_size)] for \_ in range(self.num\_blocks \*\* 2)]

    def insert(self, row, col, value):

        block\_row = row // self.block\_size

        block\_col = col // self.block\_size

        inner\_row = row % self.block\_size

        inner\_col = col % self.block\_size

        self.blocks[block\_row \* self.num\_blocks + block\_col][inner\_row][inner\_col] = value

    def display(self):

        for i in range(self.num\_blocks):

            for j in range(self.num\_blocks):

                print("Block ({}, {}):".format(i, j))

                for row in range(self.block\_size):

                    for col in range(self.block\_size):

                        global\_row = i \* self.block\_size + row

                        global\_col = j \* self.block\_size + col

                        if global\_row < self.size and global\_col < self.size:

                            print(self.blocks[i \* self.num\_blocks + j][row][col], end=" ")

                        else:

                            print("0", end=" ")

                    print()

                print()

def conformal\_decomposition(matrix):

     length = len(matrix)

     if length != len(matrix[0]):

          print(f"Invalid Dimension Error: Expected a square matrix but a Matrix({length}x{len(matrix[0])}) was given.")

          return None, None

     diamat = [[0]\*length for i in range(length)]

     offdiamat = [[0]\*length for i in range(length)]

     for i in range(length):

          for j in range(length):

               if i == j:

                    diamat[i][j] = matrix[i][j]

               else:

                    offdiamat[i][j] = matrix[i][j]

     return diamat, offdiamat

matrix = BlockMatrix(4, 2)

value = 1

for i in range(4):

    for j in range(4):

        matrix.insert(i, j, value)

        value += 1

print("Block Matrix:")

matrix.display()

mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print("Conformal Decomposition matrix:")

for i in mat:

     for j in i:

          print("{:>5}".format(j), end="")

     print()

diamat, offdiamat = conformal\_decomposition(mat)

print("Diagonal Matrix: ")

for i in diamat:

     for j in i:

          print("{:>5}".format(j), end="")

     print()

print("Off-Diagonal Matrix: ")

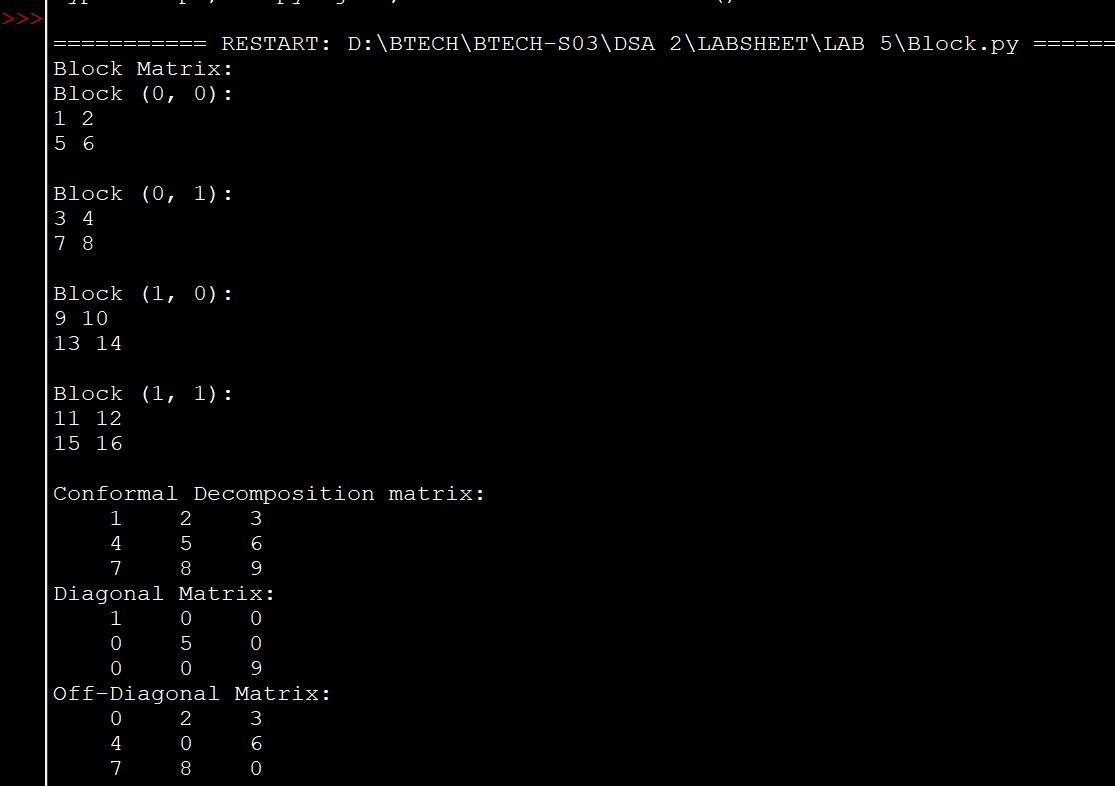
for i in offdiamat:

     for j in i:

          print("{:>5}".format(j), end="")

     print()

**Output :**

****